Using MMX™ Instructions to Implement a SchurWeiner Filter

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

1.0. INTRODUCTION
2.0. THE SCHUR ALGORITHM
    2.1. Input and Output Data Representation
    2.2. Error Correction Techniques
3.0. IMPLEMENTING THE SCHUR ALGORITHM USING MMX™ INSTRUCTIONS
    3.1. Analyzing the Schur Algorithm
    3.2. Creating the Initial Generator Matrix
    3.3. Calculating the mth Order Reflection Coefficient
    3.4. The Inner Loop: Recalculating the Generator Matrix
4.0. CODE PERFORMANCE
APPENDIX A: 'C' Version of the Schur Algorithm
APPENDIX B: The MMX™ Technology Version of the Levinson-Durbin Algorithm

1.0. INTRODUCTION

The Intel Architecture (IA) media extension includes single-instruction, multi-data (SIMD) instructions. This application note presents examples of code that exploits these instructions. Specifically, the Schur function presented here illustrates how to use the multiply and add instruction (PMADDWD>) to perform multiplication more efficiently. The performance improvement relative to traditional IA code is a result of the ability to efficiently perform multiple multiply operations in fewer cycles. To perform a signed Q15 multiply would use the IA IMUL> instruction and would take as many as 10 cycles. Using the PMADDWD> instruction, you can perform four word multiplies and two DWORD adds in three cycles. The performance gain can also be attributed to the fact that these new instructions operate on packed 64-bit values instead of 32-bit values.

2.0. THE SCHUR ALGORITHM

The amount of data which represents a human voice or sound is most often too large to store on a typical PC. Therefore, encoding the sound and only storing a partial set of the data is more practical. Voice encoding is one of the applications for which the Schur algorithm is used. This algorithm generates a set of reflection coefficients in a recursive manner as follows:

Step 1. Initialize the generator matrix *G[2][pSize+1]* with elements from the input vector *r[pSize+1]* as follows:

0 r(1) r(2) r(3) ..... r(p)
r(0) r(1) r(2) r(3) ..... r(p)

Step 2. Shift the second row of the generator matrix to the right by one place and discard the last element.

0 r(1) r(2) r(3) ..... r(p)
0 r(0) r(1) r(2) ..... r(p-1)

Step 3. Initialize *m* to 1

Step 4. Calculate the reflection coefficient *K[m]* as follows:

K[m] = - G[0][m] / G[1][m]

Step 5. Multiply the generator matrix by the following matrix:

1 K(m)
K(m) 1

Step 6. Shift the second row of the resultant matrix of step 5 by one place to the right to form the new generator matrix.

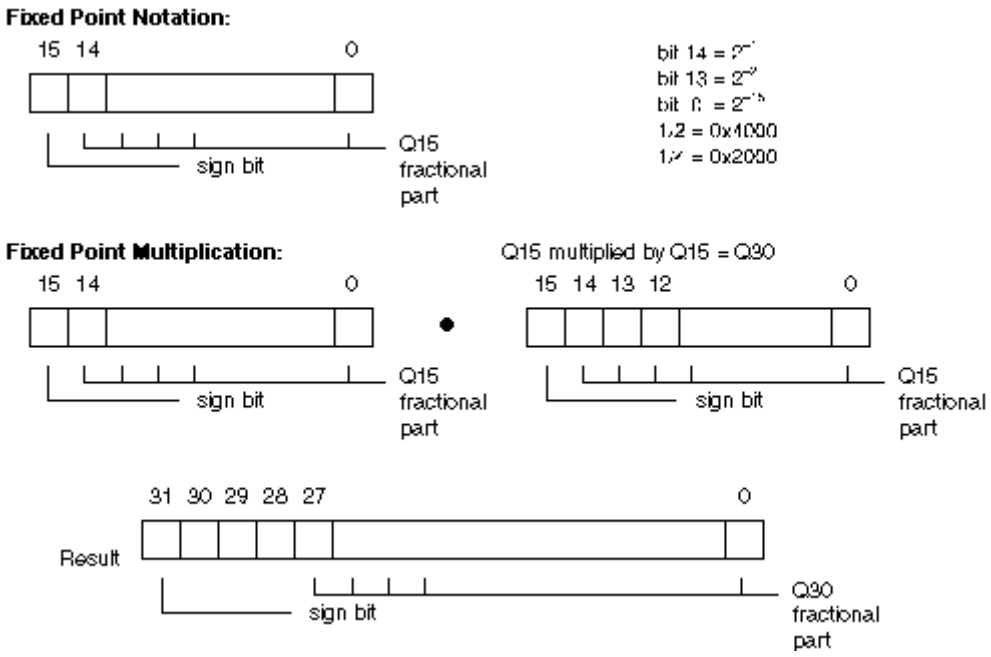
Step 7. Increment *m*.

Step 8. Repeat steps 4 through 7 until all pSize reflection coefficients have been solved.

2.1. Input and Output Data Representation

The input and output of this version of the Schur algorithm is represented in fixed-point notation. The input vector *r[ ]* and the resultant reflection coefficients *K[ ]* are represented as Q15 fractions stored in an array of short integers of size *pSize+1*. Throughout this algorithm multiplication and division of Q fractions are performed. It is important for the reader to understand what the resultant precision of such operations will be. If two signed Q15 fractions are multiplied together, the result will be a Q30 signed fraction. To determine the number of bits assigned to the fractional part of the result of a multiplication, add the number of bits assigned to the fractional parts of each multiplicand. To determine the number of bits assigned to the fractional part of the result of a division, subtract the number of bits assigned to the denominator from the number of bits assigned to the numerator. Refer to Figure 1 for further clarification.

Figure 1. Fixed-Point Notation



2.2. Error Correction Techniques

There are various areas in this algorithm where errors can occur. These errors are due primarily by the use of integer instead of floating-point arithmetic. This version of the Schur algorithm performs one error correction technique to minimize the overall amount of error found in the output results. It is accomplished by rounding results to the nearest digit of precision prior to converting from one precision to another (i.e., *Q30* to *Q15*). Refer to Example 1 for a graphical representation of fixed-point conversion techniques.

Example 1. Fixed-Point Conversion Technique

```
Converting from a Q30 to a Q15 signed fraction:
       0x2A234238     Q30 signed fraction
       + 0x00004000   rounding factor

       0x2A238238     rounded result
>> 15
       0x00005447     conversion result
```

3.0. IMPLEMENTING THE SCHUR ALGORITHM USING MMX™ INSTRUCTIONS

There were numerous areas within the Schur algorithm which could be easily adapted and optimized for use with the MMX instruction set. There was also one area which could not. The remainder of this section discusses each part of the Schur algorithm and the optimizations used during programming.

3.1. Analyzing the Schur Algorithm

Prior to implementing the code, the steps of the Schur algorithm were analyzed to understand data organization and MMX instruction set pitfalls. This analysis was used to determine the approach to take when implementing the assembly version. During the analysis, four major pitfalls were found.

The first pitfall noted was the initialization of the generator matrix. This initialization entails a series of data moves from one memory location to another. If the data resides in the data cache, the accesses can be completed in one cycle if the access is aligned. If the data does not reside in the data cache, the speed of the memory access is bound by both the processor performance and the system implementation.

The second pitfall noted pertained to the integer division. This division of *G[0][m]* by *G[1][m]* can not be avoided since the denominator changes with each iteration of the main loop.

The third pitfall noted was the shifting right of the second row of the generator matrix by one element. This action is done in steps 2 and 6 and entails moving data from one memory location to another. If this step could be avoided many cycles could be saved.

The last pitfall noted was the data organization of the generator matrix with respect to step 5 of the Schur algorithm. To increase the performance of Step 5 (the inner loop), it would be optimal to increase the number of multiplications which could be done with each iteration of the loop. For instance, the new instruction set contains an instruction called PMADDWD which can perform up to four word multiplies and two DWORD adds in three cycles. To perform multiple multiplies, it is most likely that reading more than one element at a time from the generator matrix would be necessary. Therefore, it is optimal to align data accesses (longer than a word) to the generator matrix. For example, assume that the generator matrix is aligned on a quadword boundary. With the first iteration of the main loop, data accesses to the first row of the generator matrix would be from the second element through the last element. If a doubleword or quadword access was used, each access would be unaligned and would incur an unaligned penalty of three cycles.

3.2. Creating the Initial Generator Matrix

Creating the initial generator matrix consists of a series of data moves from one memory location to another. As mentioned previously, the performance of this section of code differs depending on if the data is present in the data cache. Assuming the data is not in the data cache, this step is bound by the amount of data accesses generated. Therefore, reducing the number of data accesses will increase the performance of this section of code. To reduce the number of data accesses, it is necessary to read and write as many data elements as possible with each memory access. The new instruction set supplies the quadword move instruction MOVQ. This instruction was used to read four input data elements at one time from the input

vector r and to write four data elements at one time to the generator matrix. Example 2 contains the assembly code for initializing the generator matrix.

Example 2. Initializing the Generator Matrix

```
initialize_generator_matrix:
        movq    mm0, [eax][edx*2]                       ; get the next quadword
                                                                ; of rMatrix
        movq    [esi][edx*2], mm0                       ; store it in G0Matrix
        movq    [ebx][edx*2], mm0                       ; store it again in G1Matrix
        add     edx, 4                                  ; increment the loop counter
                                                                ; by 4 elements
        cmp     edx, ecx                                ; is i <= pSize?
        jle     initialize_generator_matrix    ; if not then exit the loop
```

Note that according to the algorithm, the first element of the generator matrix is initialized to zero. During the analysis of the algorithm, it was found that this first element is never used. Therefore, initializing it to the first element of the input vector is not harmful.

Also note that since quadword data accesses are used, the amount of space allocated to the input vector and to each row of the generator matrix must be divisible by four words. This data restriction was chosen to decrease the amount of checking overhead for this loop.

3.3. Calculating the mth Order Reflection Coefficient

The m^{th} order reflection coefficients are calculated by dividing *G[0][m]* by *G[1][0]*. This integer division can not be avoided since the denominator changes for each iteration of the main loop. It takes 43 clock cycles every time that it is executed. If the denominator value remained constant with each iteration of the outer loop, one divide could have been executed prior to entering the main loop to determine *1/denominator* constant. This value could then be used in an integer multiplication using the IMUL instruction and would take 10 cycles to execute. The IMUL instruction would have to be used in this instance since the new instruction set does not supply a doubleword multiply instruction. A floating-point divide could also be executed in place of the fixed-point integer divide but would incur an ~100 clock cycle penalty of switching from floating-point code to the MMX code. Therefore it was not used.

Notice that in the division *G[1][0]* is used and not *G[1][m]*. In the analysis, we stated that if shifting of the second row of the generator matrix can be avoided many cycles would be saved. This shifting was avoided and will be explained further in the next section.

3.4. The Inner Loop: Recalculating the Generator Matrix

With each iteration of the main loop, the generator matrix must be recalculated based on the new *K(m)*. The performance of this inner loop is critical to the overall performance of the code and extra time was spent in optimizing this loop. Below in Example 3 is the C language code which recalculates the generator matrix following the steps of the algorithm. This code will give the user a better idea of how this section of the algorithm is implemented.

Example 3. Recalculating the Generator Matrix (C code)

```
/*******************************************************
 * Recalculate the generator matrix entries based on    *
 * the new reflection coefficient K[m].                 *
 *******************************************************/
```

```
        for ( i = m ; i <= p; i++)
                {
                /* perform the matrix multiplication of each
                 * element
                 */
                temp1 = ((G[0][i] * K[i]) + 0x4000) >> 15;
                temp2 = ((G[1][i] * K[i]) + 0x4000) >> 15;
                /* perform the addition and store the results
                 */
                G[0][i] = G[0][i] + temp2;
                G[1][i] = G[1][i] + temp1;
                }
/*******************************************************
 * Shift the second row over by one element.           *
 *******************************************************/
        for ( i = p; i > m; i--)
                G[1][i] = G[1][i-1];
```

One performance gain mentioned in Section 3.1 was to decrease the number of unaligned data accesses to the generator matrix. To accomplish this, the shifting of the second row of the generator matrix was not performed. The algorithm was slightly changed to take this into consideration. By avoiding the shift and ensuring that the second row of the generator matrix is doubleword aligned, the number of unaligned accesses are decreased by half. Figure 2 depicts the change made to the algorithm.

Figure 2. Generator Matrix Usage

Initial Generator Matrix Setup:
G[0]
G[1]
*Calculate K[1] = G[0][1] / G[1][0] = A*0 / a0 where m = 1

Recalculate the Generator Matrix
G[0]
G[1]
Calculate K[2] = G[0][2] / G[1][0] = B1/ a1 where m = 2

Recalculate the Generator Matrix
G[0]
G[1]
Calculate K[3] = G[0][3] / G[1][0] = C 2/a 2 where m = 3

Recalculate the Generator Matrix
G[0]
G[1]
Calculate K[4] = G[0][4] / G[1][0] = D3/a3 where m = 4

where: m is the main loop counter

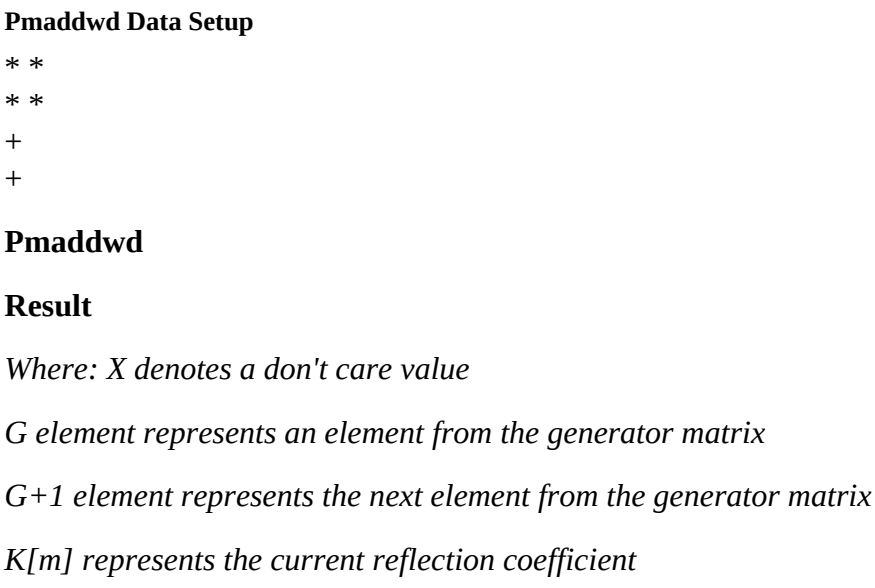
P is the number of reflection coefficients to solve for

G[0] is the first row of the generator matrix

G[1] is the second row of the generator matrix

Another performance gain which was mentioned in Section 3.1 was to perform as many multiplications with one instruction as possible. The new MMX instruction set contains two types of multiply instructions: PMADD and PMUL. The data must be analyzed to determine which instruction to use and how many multiplies can be performed with one instruction. Each multiplicand is a *Q15* number. The result of each multiply is a *Q30* number. The PMULHW could be used to perform four word multiplies but would prevent data rounding prior to the final conversion from *Q30* to *Q15*. The PMADDWD instruction could be used but would only be able to perform two word multiplies. Using the PMADDWD instruction would retain data accuracy and allow for rounding of the result prior to performing the final data conversion from *Q30* to *Q15*. For this implementation, the PMADDWD> instruction was chosen so that the simple error correction technique discussed in Section 2.2 could be applied to minimize the overall error introduced by the use of integer arithmetic. Below in Figure 3 is a representation of the data setup used to perform the multiplications.

Figure 3. Data Setup for the Inner Loop



Because of the nature of the algorithm, four multiplications at a minimum must be performed with each iteration of the loop (two for the first row and two for the second). This is necessary because the new values of the first row are dependent on the previous values of the second row. Also, the new values of the second row are dependent on the previous values of the first row. For example, the new value of *G[0][m]* is calculated *as G[0][m] + (G[1][0] * K[m])* and the new value of *G[1][0]* is calculated *as G[1][0] + (G[0][m] * K[m]).* Therefore, if two calculations are performed for the first row, it is necessary to perform two calculations for the second row.

The assembly code which was written to implement the recalculation of the generator matrix can be seen in Example 4. This code takes into consideration the previously mentioned optimizations. Assuming all data is in the data cache, this code takes 16 cycles per iteration to execute when accesses to the first row of the generator matrix are aligned. Since four multiplies are performed with each iteration of the inner loop, it takes four cycles to calculate each new generator matrix element. When accesses to the first row of the generator matrix are unaligned, the code takes 19 cycles per iteration and takes

Example 4. The Inner Loop

```
_inner_loop:
1.      cmp             edx, eax                        ; is m > pSize
2.      jg              _inner_loop_done                ; if so, jump out of this loop
3.      movd            mm0, [esi][edx*2]               ; get G[0][m] and G[0][m+1]
4.
5.      movd            mm2, [ecx]                      ; get G[1][i] and G[1][i+1]
6.      punpcklwd       mm0, mm0                        ; mm0 = G[0][m+1], G[0][m+1],
G[0][m],
                                                        ;         G[0][m]
7.      movq            mm5, mm0                        ; make a copy of the G[0] data
setup
8.      punpcklwd       mm2, mm2                        ; mm2 = G[1][i+1], G[1][i+1],
G[1][i],
                                                        ;       G[1][i]
9.      movq            mm6, mm2                        ; make a copy of the G[1] data
setup
10.     pmaddwd mm5, mm7                        ; Calculate G[0][m+1] * K[m],
                                                        ; G[0][m] * K[m]
11.     psrad           mm2, 16                 ; Sign extend G[1][i+1] and G[1][i]
12.     pmaddwd mm6, mm7                        ; Calculate G[1][i+1] * K[m],
                                                        ; G[1][i] * K[m]
13.                                                     ; lose a cycle here since you
must
14.                                                     ; wait three cycles after a
pmaddwd prior
                                                        ; to using its results
15.     psrad           mm0, 16                 ; Sign extend G[0][m+1] and G[0][m]
16.     paddd           mm5, mm1                        ; Add the rounding factor prior
to
                                                        ; converting from Q30 to Q15
17.     psrad           mm5, 15                 ; convert from Q30 to Q15
18.     paddd           mm6, mm1                        ; Add the rounding factor prior
to
                                                        ; converting from Q30 to Q15
19.     psrad           mm6, 15                 ; convert from Q30 to Q15
20.     paddd           mm2, mm5                        ; Calculate G[1][i+1] +
G[0][m+1]*K[m]
                                                        ; and G[1][i] + G[0][m]*K[m]
21.     packssw mm2, mm2                        ; Convert from double to packed words
22.     paddd           mm0, mm6                        ; Calculate G[0][m+1] +
G[1][i+1]*K[m]
                                                        ; and G[0][m] + G[1][i]*K[m]
23.                                                     ; lose a cycle here since you
must wait
24.                                                     ; 1 cycle after modifying a
register
                                                        ; prior to storing it.
25.     movd            [ecx], mm2                      ; Store the G[1] results
26.     packssdw        mm0, mm0                        ; Convert from double to packed
words
27.     add             ecx, 4                  ; Increment the G[1] pointer
28.                                                     ; This won't pair with the next
                                                        ; instruction since memory
operations
```

```
                                                    ; do not pair with instructions
which
                                                    ; operate on integer registers.
29.     movd            [esi][edx*2], mm0           ; Store the G[0] results
30.
31.     add             edx, 2                  ; Increment the loop counter by 2
32.     jmp             _inner_loop                 ; Jump to the start of the
inner loop
```

There are two areas in the inner loop where instruction penalties occur. The first penalty occurs on line 16. This instruction uses MM5 as a source. This register contains the result of the PMADDWD> instruction issued on line 10. Since three cycles have not elapsed from the time the PMADDWD instruction was issued to the time the result was used as a source, a one cycle penalty occurs. The second penalty occurs on line 25. This instruction stores the value in MM2 into the location pointed to by the register ECX. The register MM2 was modified in the instruction prior to the store. Since two cycles have not elapsed from the time MM2 was used to the time it was stored to memory, a one cycle penalty occurs.

4.0. CODE PERFORMANCE

The overall performance of this version of the Schur Weiner Filter is based on two sections of the code: the calculation of *K[m]* and the recalculation of the generator matrix. The following discussion assumes all data is present in the data cache and that all data is aligned.

To calculate *K[m]* and to set up register values prior to re-calculating the generator matrix takes approximately 55 clock cycles. The majority of these cycles can be directly attributed to the integer divide which executes in 43 cycles. As mentioned previously, this integer divide can not be avoided.

There are two cycle count numbers which are important to take into consideration when approximating the number of cycles it takes to recalculate the generator matrix. When all accesses to the first row of the generator matrix are aligned, it takes 16 clock cycles per iteration to recalculate four generator matrix elements. When all accesses to the first row of the generator matrix are unaligned, it takes 19 clock cycles per iteration to recalculate 4 generator matrix elements. Since half of the accesses to the first row of the generator matrix are aligned and half are unaligned, on average it takes 4.375 cycles to recalculate each element.

Applying these cycle count numbers to an application which calculates *pSize* reflection coefficients using an input vector of size *pSize+1,* an approximate overall performance can be obtained as follows:

$$(55 * pSize) + \sum_{m=0}^{pSize-2} (2)(4.375)(pSize - m)$$

For the C language version of the Schur algorithm, refer to Appendix A. For the complete assembly version of the Schur algorithm refer to Appendix B. The assembly version is reentrant but the user may wish to dispose of the local arrays and pass them in as pointers to a heap space location if stack usage is an issue.

APPENDIX A: "C' Version of the Schur Algorithm

```
* Description:

schur_int is the C language version of the Schur
algorithm used to calculate the

reflection coefficients of a given  set of linear
equations. The input matrix

is assumed to be a set of 15-bit signed fixed-point
short integers. The resultant

reflection coefficients will also be 15-bit signed
fixed-point short integers.

 * Input:

r short int *  the input matrix representing the
set of linear

equations of which to find the reflection

coefficients. Each array element must be a

Q15 short integer.

K short int *  the output matrix containing the
resultant

reflection coefficients represented as Q15

short integers.

p int  the number of reflection coefficients to

solve for (typically 10 or 16).


void Schur (short *r, short *K, int  p )

{   // begin Schur()

short i,     // generator matrix index

      m;    // order counter

short G[2][16];   // generator matrix; the number

    // of columns (16) was determined

    // by the input size.

 // initialize the generator matrix

 for (i = 0; i < p; i++)
```

```
  {   // begin for loop

  G[0][i] = r[i+1];

  G[1][i] = r[i];

  }   // end for loop

 // for each order,  calculate the new reflection
coefficient

 // and the new generator matrix for the next order

 for (m = 1; m <= p; m++)

  {    // begin for loop

  // calculate k[m].  Since the result should be
a Q15 number,  the

  // division should be between a Q30 and a Q15 number.
 Therefore

  // convert the numerator to a Q30 number prior
to performing the

  // divide.

  K[m] = -(short)(((long)G[0][m-1] << 15) /
((long)G[1][m-1]));

  // calculate the new generator matrix.  Error correction
techniques

  // must be used during this section.  When multiplying
2 Q15 numbers

  // the result will become a Q30 number.  This number
must then be

  // converted back to a Q15 number.  To performing
the rounding

  // error correction technique,  add 0x4000 to the
Q30 number prior

  // to shifting it right by 15 bits.

  for (i = p-1; i >= m; i--)

   {   // begin for loop

   G[0][i] = (short)((((long)G[0][i] << 15)
+ (G[1][i] * K[m]) + 0x4000)

    >> 15);
```

```
   G[1][i] = (short)((((long)G[1][i-1] << 15)
+ (G[0][i-1] * K[m]) +

    0x4000) >> 15);

   }   // end for loop

  }   // end for loop

}   // end schur_int()
```

APPENDIX B: The MMX™ Technology Version of the Levinson-Durbin Algorithm

```
;* Description:

;* The purpose of this file is to provide the MMX
code for the  schur algorithm as

;* an instructional example to those who are just
beginning to code using the new

;* instructions.

;*

;* Assumptions:

;* 1.     The set of normal equations given are stable

;* 2.     The normal equations are represented by
one matrix 'r' which

;*         contains the coefficients gamma(0) through
gamma(p) given as

;*         15-bit signed fixed-point short integers
(Q15).

;* 3.     The number of short integers allocated
for matrix 'r' is

;*         divisible by 4.  Used elements are initialized
to 0.

;* 4.     The resultant reflection coefficients will
be returned as

;*         15-bit signed fixed-point short integers
(Q15).

 TITLE Schur

 .486P

.model FLAT

;*****************************************************************************

;*     TEXT SEGMENT

;*****************************************************************************

_TEXT SEGMENT
```

```
;

; Declare schur as a public routine to allow the
'C' code to

; call it.

;

PUBLIC schur


;* Description:

;* schur is the optimized  MMX technology  version
of the Schur

;* algorithm.  It is used to calculate the reflection
coefficients of a

;* given set of  normal equations. The steps of the
algorithm are as

;* follows:

;*  Step 1:  Initialize the generator matrix to

;*    G0 =    0,    r(1), r(2), r(3), .... r(p)

;*    G1 =    r(0), r(1), r(2), r(3), .... r(p)

;*  Step 2:  Shift the elements in G1 by 1 position
to obtain:

;*    G0 =    0,    r(1), r(2), r(3), .... r(p)

;*    G1 =    0,    r(0), r(1), r(2), .... r(p-1)

;*  Step 3:  Initialize m to 1

;*  Step 4:  Calculate K[m] as follows

;*    K[m] = - G0(m) / G1(m)

;*  Step 5:  Multiply the generator matrix by the
matrix V where

;*    V is defined as:

;*    V = 1      K(m)

;*        K(m)   1

;*  Step 6:  Shift the elements in G1 by 1 position

;*  Step 7:  Repeat steps 4 through 6 for each reflection
```

```
;*    coefficient that needs to be calculated
;*
;* Inputs:
;* rMatrix short int *  a pointer to the first element
;*    of the input 'r' matrix
;* kMatrix short int *  a pointer to the output
;*    reflection coefficients array
;* pSize short int  the number of reflection coefficients
;*    to solve for (typically 10 or 16).

schur PROC C USES ebx ecx edx esi,
rMatrix:PTR WORD,
kMatrix:PTR WORD,
pSize:DWORD
;
; Declare local variables
;  1. G0Matrix     - row 1 of the generator matrix
;  2. G1Matrix     - row 2 of the generator matrix
;  3. mSave        - used to store the loop counter
'm'
;  4. round_factor - contains the rounding factor
0x4000
;
 LOCAL G0Matrix[72]:WORD
 LOCAL G1Matrix[72]:WORD
 LOCAL mSave:DWORD
 LOCAL round_factor[2]:DWORD
;
; Initialize local variables
 mov  round_factor, 04000H
```

```
 mov  round_factor + 4, 04000H

;

; Setup registers in preparation to initialize the

; generator matrix G0 and G1

;

 mov  eax, rMatrix   ; get the address of rMatrix

 lea  esi, G0Matrix  ; get the address of G0Matrix


 mov  ecx, pSize   ; get pSize

 mov  edx, 0

 lea  ebx, G1Matrix  ; get the address of G1Matrix


;

; Initialize the generator matrix G0 and G1

;

initialize_generator_matrix:

 movq  mm0, [eax][edx*2]  ; get the next quadword

      ; of rMatrix

 movq  [esi][edx*2], mm0  ; store it in G0Matrix


 movq  [ebx][edx*2], mm0  ; store it again in G1Matrix


 add  edx, 4   ; increment the loop counter

      ; by 4 elements

 cmp  edx, ecx   ; is i <= pSize?

 jle  initialize_generator_matrix ; if not then exit
the loop

;

; if (m > pSize) then we are done!

;

 mov  edx, 1   ; initialize m
```

```
 cmp  edx, ecx   ; is m > pSize?

_schur_main_loop:

;

; This is the main loop.

; Setup the numerator and denominator values for
the calculation of K[m].

; The numerator must be converted to a Q30 number
and the denominator will

; remain a Q15 number so that when the division is
completed the result will

; be a Q15 number.

;

 mov  ax, [esi][edx*2]  ; get the numerator value

 jg  _schur_main_loop_done ; exit the main loop if
m > pSize

 shl  eax, 16   ; convert from Q15 to Q30

 lea  ecx, G1Matrix  ; get the address of G1Matrix


 sar  eax, 1   ; finish the numerator conversion

 mov  ebx, [ecx]   ; get the denominator value

 shl  ebx, 16   ; sign extend the denominator

 mov  mSave, edx  ; save m

 sar  ebx, 16   ; finish the sign extension

 mov  edx, eax   ; copy the numerator to edx

 sar  edx, 31   ; fill edx with the sign extension

 mov  ecx, 0

 idiv  ebx   ; calculate K[m]

 mov  ebx, kMatrix  ; get the address of kMatrix

 sub  ecx, eax   ; negate K[m]


 mov  edx, mSave  ; get the loop counter 'm'
```

```
 pxor  mm4, mm4  ; clear out mm4

 movd  mm7, ecx   ; setup mm7 with 0, K[m], 0, K[m]


 movq  mm1, round_factor  ; load the rounding factor
into mm1

 punpcklwd mm7, mm4  ; continue setting up mm7

 mov  [ebx][edx*2], ecx  ; store K[m] in kMatrix

 lea  ecx, G1Matrix  ; get the address of G1Matrix


 mov  eax, pSize   ; get the value of pSize

 punpckldq  mm7, mm7  ; finish setting up mm7

;

; This is the inner loop.

; Recalculate the generator matrix using the following
algorithm

; for (i = 0; m <= pSize; i = i+2, m = m+2)

;  {

;  temp1 = (short int)((G0(m) * K(m) + rounding_factor)
>> 15);

;  temp2 = (short int)((G0(m+1) * K(m) + rounding_factor)
>> 15);

;  temp3 = (short int)((G1(i) * K(m) + rounding_factor)
>> 15);

;  temp4 = (short int)((G1(i+1) * K(m) + rounding_factor)
>> 15);

;  G0(m) = G0(m) + temp3;

;  G0(m+1) = G0(m+1) + temp4;

;  G1(i) = G1(i) + temp1;

;  G1(i+1) = G1(i+1) + temp2;

;  }

;

_inner_loop:
```

```
 cmp  edx, eax   ; is m > pSize

 jg  _inner_loop_done  ; if so, jump out of this
loop

 movd  mm0, [esi][edx*2]  ; get G0(m) and G0(m+1)


 movd  mm2, [ecx]  ; get G1(i) and G1(i+1)

 punpcklwd mm0, mm0  ; mm0 = G0(m+1), G0(m+1), G0(m),
G0(m)

 punpcklwd mm2, mm2  ; mm2 = G1(i+1), G1(i+1), G1(i),
G1(i)

 movq  mm5, mm0  ; make a copy of the G0 data setup


 movq  mm6, mm2  ; make a copy of the G1 data setup

 pmaddwd  mm5, mm7  ; Calculate G0(m+1) * K(m), G0(m)
* K(m)

 pmaddwd  mm6, mm7  ; Calculate G1(i+1) * K(m), G1(i)
* K(m)

 psrad  mm2, 16   ; Sign extend G1(i+1) and G1(i)
to DWORDs

 psrad  mm0, 16   ; Sign extend G0(m+1) and G0(m)
to DWORDs

 paddd  mm5, mm1  ; Add the rounding factor to G0(m+1)
* K(m)

      ; and G0(m) * K(m)

 psrad  mm5, 15   ; Convert G0(m+1) * K(m), G0(m)
* K(m) to Q15

 paddd  mm6, mm1  ; Add the rounding factor to G1(i+1)
* K(m)

      ; and G1(i) * K(m)

 psrad  mm6, 15   ; Convert G1(i+1) * K(m), G1(i)
* K(m) to Q15

 paddd  mm2, mm5  ; Calculate G1(i+1) + G0(m+1) *
K(m) and

      ; G1(i) + G0(m) * K(m)

 packssdw  mm2, mm2  ; Convert the two DWORD word
results to packed words
```

```
 paddd  mm0, mm6  ; Calculate G0(m+1) + G1(i+1) *
K(m) and

      ; G0(m) + G1(i) * K(m)

 movd  [ecx], mm2  ; Store the G1 results

 packssdw  mm0, mm0  ; Convert the two DWORD word
results to packed words

 add  ecx, 4   ; Increment the address of G1Matrix


 movd  [esi][edx*2], mm0  ; Store the G0 results


 add  edx, 2   ; Increment m by 2

 jmp  _inner_loop  ; Jump to the start of the inner
loop

;

; Increment the main loop counter m and compare it
against pSize.

;

_inner_loop_done:

 mov  edx, mSave  ; Get the current value of m

 inc  edx   ; Increment m

 mov  ecx, pSize   ; Get the value of pSize

 cmp  edx, ecx   ; is m > pSize?

 jmp  _schur_main_loop  ; Jump to the start of the
main loop

;

; We are done so it's time to return!

;

_schur_main_loop_done:

 emms     ; flush floating-point stack

 ret  0   ; Return to the calling procedure

schur ENDP
_TEXT ENDS
END
```